Edicts: Implementing Features with Flexible Binding Times

Venkat Chakravarthy John Regehr Eric Eide
University of Utah, School of Computing
{vchakra, regehr, eeide}@cs.utah.edu

Abstract

In a software product line, the *binding time* of a feature is the time at which one decides to include or exclude a feature from a product. Typical binding site implementations are intended to support a single binding time only, e.g., compile time or run time. Sometimes, however, a product line must support features with variable binding times. For instance, a product line may need to include both embedded system configurations, in which features are selected and optimized early, and desktop configurations, in which client programs choose features on demand.

We present a new technique for implementing the binding sites of features that require flexible binding times. Our technique combines design patterns and aspect-oriented programming: a pattern encapsulates the variation point, and targeted aspects—called *edicts*—set the binding times of the pattern participants. We describe our approach and demonstrate its usefulness by creating a middleware product line capable of serving the desktop and embedded domains. Our product line is based on JacORB, a middleware platform with many dynamically configurable features. By using edicts to select features at compile time, we create a version of JacORB more suited to resource-constrained environments. By configuring four JacORB subsystems via edicts, we achieve a 32.2% reduction in code size. Our examples show that our technique effectively modularizes binding-time concerns, supporting both compile-time optimization and run-time flexibility as needed.

1. Introduction

Software product lines are an attractive solution for software development because they are economical and adaptable to change. A software product line, as defined by Clements and Northrop [2], is "a set of software-intensive systems that share a common, managed set of features... and that are developed from a common set of core assets in a prescribed way." The members of a product line—i.e., individual products—differ from each other according to their sets of selected features.

The selection of features in a product can occur at different times, as required by many development issues [22]. For example, consider the input-handling feature for text-processing software on a device like a cell phone. If the device supports only a keypad, then there is only one possible method of input. In this case, the keypad input feature can be selected *early*—e.g., by the system builder, when the software for the phone is compiled. However, one can also imagine advanced handsets that support interactive input through a combination of a keypad, a touch-screen stylus, and voice. In this situation, the selection of a specific input handler must occur *late*—e.g., at run time—based on the input mode the user chooses. A company may want to develop a single software product line that supports both cases described above. That is, it would like to provide two cell-phone software systems that have many features in common (both support keypad input) but which differ in *when* those features are selected. It would be convenient if the software developers could easily choose the feature selection times when configuring their software for different cell phone models.

In product line terminology, *variation points* control the inclusion or exclusion of features within the products that are part of a product line. A *binding site* is the realization of a variation point in the implementation; in general, a single conceptual variation point may translate to numerous (scattered) binding sites in the code. *Binding time* refers to the time at which the decisions for a variation point are set [5]: in other words, it is the time at which feature selection occurs. There are many different binding times available to a software designer—e.g., compile time, link time, and run time. A software development technique that is used to implement a variation point is called a *variability mechanism* [12].

Configuration scripts, conditional compilation, static and dynamically linked libraries, virtual dispatch tables, reflection, and dynamic class loading are examples of mechanisms that can be used to implement variation points [7, 8]. Each of these, however, is strongly tied to a particular choice of binding time. When one of these mechanisms is used to bind a feature, any subsequent need to change the binding time means replacing the mechanism: that is, modifying the source code of every binding site. In a continuously evolving product line, where binding times may change based on existing, evolving, or expanding domain requirements, this process is error-prone and the code modifications are tedious to track. Indeed, others have previously identified binding-time flexibility as an important concern for product-line software [23].

In this paper, we describe a novel technique to provide binding-time flexibility in a modular and convenient manner. Our variability mechanism makes it possible to choose between compile-time and run-time binding for selected features. In general, earlier binding times enable better static analysis and system optimization, whereas later binding times enable configuration by users and post-deployment updates. Compile-time binding is therefore often appropriate for constrained systems, where constraints may be set by the purpose or operating environment of the software. For example, because embedded systems have limited memories and real-time deadlines, such systems may only support a subset of the features available in a product line. Conversely, desktop and server systems can easily support large and feature-rich software packages, and so may favor run-time binding.

This material is based upon work supported by the National Science Foundation under Grant No. 0410285.

The definitive version was published in *Proceedings of the Seventh International Conference on Aspect-Oriented Software Development (AOSD)*, Brussels, Belgium, Mar.–Apr. 2008, http://doi.acm.org/10.1145/1353482.1353496

The variability mechanism we propose uses design patterns and aspect-oriented programming in combination to achieve binding-time flexibility. First, a programmer encapsulates a variation point within the implementation of a design pattern [9]. The pattern serves to make the point identifiable and stable, and it provides the "hooks" that are necessary for changing the point's binding time. The programmer then writes *edicts* to manipulate the binding times at such points. An edict is a targeted aspect that enables a particular binding time. At the time of product assembly (i.e., when the parts of a product are selected, collected, and configured to work together), a product assembler uses knowledge about the domain and chooses edicts to implement the desired feature-binding strategies. The assembled system can be statically optimized, based on information obtained from the chosen edicts.

We demonstrate and evaluate our technique by creating a small product line from an existing middleware system called JacORB [13]. The original JacORB is designed to run on PC-class (Java SE) devices and has many dynamically configurable features. Our experiment was to extend JacORB to be suitable for smaller, handheld-class devices (Java ME-CDC platforms) as well. By reimplementing some variation points using edicts, we made it possible to create both "desktop" and "embedded" versions of the middleware from a single code base. In the desktop configuration, features are still bound at run time as in the original software. In the embedded configuration, however, edicts enforce compile-time binding at key variation points—"hard-wiring" features as appropriate for a smaller, more special-purpose device.

We show that when compile-time binding is used, our technique improves the effectiveness of static program analyzers such as ProGuard [16]. We use ProGuard in several experiments to "shrink" the bytecode of both the original JacORB middleware, minimally modified for fixed feature selection, and our modified JacORB, configured with edicts that implement compile-time feature binding. The optimized, edict-based middleware is consistently smaller—up to 32.2% smaller—than the optimized, non-edict-based middleware. This savings could be further improved, we believe, if we applied our technique to more than the four variation points we have modified thus far.

The contributions of this paper are as follows:

- We present our technique—our variability mechanism—for implementing binding sites that support features with variable binding times. Handling such variability is a significant challenge in the implementation of product-line software.
- We define a systematic method for applying our technique.
- We demonstrate our approach within an example middleware product line, one designed to encompass both constrained (Java ME-CDC) and feature-rich (Java SE) platforms.
- We evaluate the benefits of our mechanism in terms of its ability to create feature-subsetted middleware products with reduced resource demands.

Our approach changes the binding time of a feature from being a design-time attribute of a product line to being an assembly-time attribute of a product. It builds upon the modularity, stability, and traceability provided by design patterns and targeted aspects, thus promoting understanding by programmers and optimization by compilers. When our technique is applicable, it can help software architects use a single code base to create a wide variety of products that satisfy the requirements of disparate domains.

2. Background: Feature Modeling

Feature modeling [4] is a technique that helps the designers of a product line identify the characteristics that are common in applications and also those that are different among the applications written for a domain. A *feature model* is an abstract representation of the variability present in a software product line. The model uses diagrammatic notations to represent various features present in the software system, their dependencies and interactions, and the software domain concerns in a structured format. It helps anticipate the addition of new or removal of old features and maintain a record of all variants in the product line. We use the notations defined by the Feature-Oriented Domain Analysis (FODA) [14] to create feature models in this work.

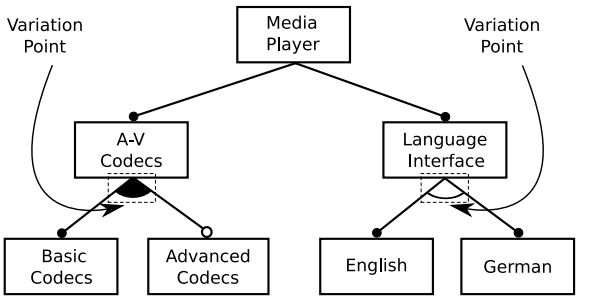

Figure 1. Hypothetical feature model for a media player

Continuing the cell phone example from Section 1, let us assume that the cell phone has media playing capabilities. We can represent the cell phone's media player subsystem using FODA notations, as shown in Figure 1. The topmost box represents the system itself, and all other boxes represent features.

Notations on edges describe constraints on how features can be combined. The filled-circle arrowheads on the edges from "Media Player" to its immediate children indicate that "A-V Codecs" and "Language Interface" are *mandatory features*—present in all media players. All players come configured with a basic set of audio and video codecs, but a deluxe model is configured with advanced codecs. This is an *optional feature* as represented by the unfilled circle arrowhead. The codecs are *or-features*: either or both may be present in the system. The filled arc below the "A-V Codecs" subsystem represents this constraint. Similarly, the language interface may be provided in English or in German but not in both. The unfilled arc below the language interface feature represents these *alternative features*.

The attachment points for optional, or-, and alternative features —where choices must be made—are called *variation points*. An important step in building a feature model is specifying when the decisions at each variation point must be set. These are the variation points' *binding times*, and typical binding times include:

- **Preprocessing time and compile time.** Preprocessor directives and compile-time features strongly influence the ways in which a product is configured and optimized.
- **Link time.** Linking can be performed at various times and depends upon the software deployment environment. It could occur immediately after compilation, for example, or even after the program starts running.
- **Initialization time.** This is a precursor phase before the program starts it normal operation. Interpreting directives from a configuration file at program startup, for example, is a common form of the initialization-time feature binding.
- **Run time.** The binding of features can occur while the program is running. Object-oriented programming languages provide a great deal of support for run-time feature selection.

Finer distinctions are possible [4]. In this paper, we are mainly interested in features that must be bound at compile time in some products and at initialization or run time in others.

3. Flexible Binding Times via Edicts

Some software product lines require variation points with flexible binding times. In this section, we describe our technique for implementing such points in a convenient and modular fashion. Our approach utilizes design patterns to encapsulate variation points in conjunction with targeted aspects, called *edicts*, that determine the binding times of features. The benefits of our technique include *configurability*, meaning that product assemblers can change binding times without rewriting source code; *traceability*, meaning that binding-time concerns are well modularized and identifiable within a system's implementation; and *specializability*, meaning that significant static optimizations can be achieved when compile-time bindings are chosen.

3.1 Overview and rationale

An edict is an aspect that implements a strategy for feature selection within the implementation of a variation point. To apply an edict, it must be possible to accurately identify the implementation of the variation point—i.e., the binding sites—that we want to affect.

To make the binding sites for a variation point apparent, we implement the sites in terms of a design pattern. In a feature model, a variation point represents a connection between two features at a high level. The nature of that connection can be complex, and often, the relationship between two features is more than can be expressed by a single programming language construct by itself. In other words, the "interface" between two features is often more than a single method or a single class [17]. In our experience, however, the interface between two features can be often (but not always) described in terms of one or more design patterns.

Design patterns are a common and popular technique for describing collections of program elements that work together [9]. Patterns are useful because they help people understand software implementations in terms of abstract but well-understood structuring concepts. They help to describe both the purpose and expected modularity of a set of program elements—e.g., the roles that are served by objects of various classes, and the ways in which the participating objects should and should not communicate with each other. Because patterns make the relationships between program elements more apparent, and at the same time communicate modularity expectations to people, design patterns help to create and preserve points of stability within a software architecture. The expressiveness and stability properties of patterns make them suitable for implementing many kinds of feature connections. Moreover, patterns are suitable for implementing many kinds of variation points—i.e., places where features are included or excluded based on decisions that are made by a software developer or user.

Typical implementations of design patterns are intended to support run-time decisions through the dynamic creation, configuration, and connection of objects. As shown in previous work, however, patterns can also be useful for implementing decisions that are set before run time, when a product is compiled or assembled [6].

To support multiple binding times most effectively, the implementation of a pattern must vary according to the times at which the participants and relationships described by the pattern become known. Although it is possible to implement static relationships via dynamic mechanisms, such techniques tend to obscure information that could be used statically to check and optimize software. Therefore, when relationships within a pattern instance are known statically, we would like to choose a pattern implementation that makes the static information immediately and readily apparent to program development and analysis tools.

Aspect-oriented programming provides the power that is needed to set or modify the implementation of a design pattern to suit the binding times of the participants. Through pointcuts and advice, an aspect can modularize both the identification and manipulation of the many classes that implement the design pattern. We can write multiple aspects that apply to a single design pattern instance (i.e., a single variation point): one aspect may implement early binding, and another late binding. We refer to these aspects as "edicts" because they represent our intent to impose—as much as possible—statically available knowledge about the domain into the design of the product. This static knowledge is the choice of a binding time and, in the case of an early-bound feature, an optimization opportunity. One can view this as a targeted, human-directed form of program specialization [3]. The intent and controlled scope of edicts distinguish them from aspects in general.

```
public aspect Edict {
    // Use ‘‘marker interfaces’’ to identify the roles
    // of the classes within the design pattern impl.
    public interface [Role] {};
    declare parents: [Class] implements [Role];

    // Use ITDs to implement pattern's methods according
    // to the desired binding strategy.
    ... [Role].[method](...) { ... }

    // Use pointcuts to identify interactions among the
    // classes of the design pattern impl.
    pointcut [pcname](...):
        [call or execution](...)
        [&& target(...)] [&& within(...)] && args(...);

    // Use ‘around’ advice to implement binding strategy
    // at points of interaction.
    ... around(...): [pcname](...) { ... }
}
```

Figure 2. The structure of a typical edict

3.2 The implementation of edicts

An edict is implemented using AspectJ aspects, and the recipe for a typical edict is shown in Figure 2. Using intertype declarations (ITDs) and around advice, an edict "fleshes out" or modifies the implementation of a variation point as necessary to enable a particular binding time. An edict may contain any number of intertype and advice declarations, as required to affect multiple sites (binding sites) within the implementation of a variation point. By implementing variation points as design patterns, as discussed above, we help to ensure that the essential join points within the implementation can be clearly identified by AspectJ pointcuts. Typically, only `call` and `execution` join points are needed to identify locations of interest, although other pointcut designators, such as `within` and `target`, are needed to restrict the edict to the implementation of a particular variation point.

Because edicts are intended to support static optimizations, they are applied when a product is compiled. For every feature controlled by edicts, the product assembler chooses the edict that implements the desired feature binding time, and he or she includes that edict in the product build.

One way to implement binding-time flexibility is to keep late (run-time) binding code intact in the "ordinary" implementation of a design pattern, and write edicts only to impose early (compile-time) binding choices via around advice. Based on our experience, however, we recommend against this approach. In early experiments, we observed that the static analysis tool we use—ProGuard—sometimes could not exploit the fact that the code for late binding had been made dead by around advice. As a result, when an early-binding edict was applied, our optimizer was unable to remove large amounts of dead code from our programs. (We describe ProGuard in more detail in Section 5.1.)

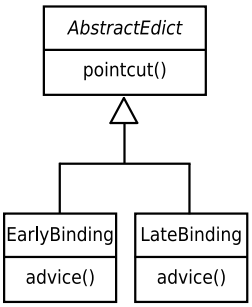

Figure 3. The hierarchy of edicts

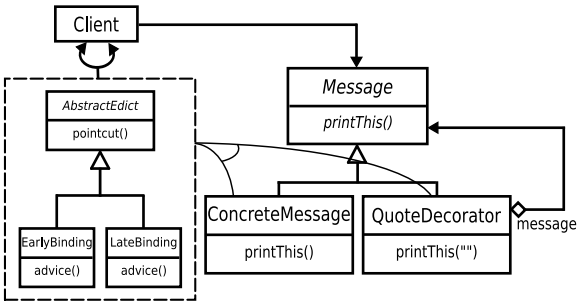

Figure 4. The structure of a message-display program

We therefore recommend writing separate edicts for run-time and compile-time bindings, as illustrated in Figure 3. A programmer encapsulates a variation point within a design pattern, but he or she leaves the parts related to binding-time decisions *unfinished* in the "base" implementation of the software. He or she then writes an abstract aspect, with pointcuts that target the binding-time decision points. The abstract aspect declares and applies marker interfaces as well. Finally, the programmer implements run-time and compile-time binding strategies as individual, concrete subaspects. Each of these contains code that completes the pattern as needed for a particular feature binding time. This approach helps to ensure that optimization opportunities are apparent to program analysis tools.

3.3 Example

To better describe edicts, we show how they could be applied to a simple example. Our example is a model of a message-display subsystem with optional features, such as one might find in a cell phone product line. The optional features control how text is displayed, e.g., in different fonts or colors. We can control the binding times of these features by using edicts in conjunction with the *Decorator* pattern.

Figure 4 shows the structure of our example program using a modified UML notation. For now, ignore the edicts on the left-hand side. The structure of the messaging program overall is based on the *Decorator* pattern, and the code of our program is shown in Figure 5. `Message` is an interface implemented by the `ConcreteMessage` class, which prints text. `QuoteDecorator` is a decorator that surrounds output strings with quotes. The main program dynamically chooses whether or not to apply the decorator, and then prints a string.

The quoting of the output message is the optional feature, and currently, this feature is bound at run time. We want to modify the program so that we can make a compile-time decision to (1) enable or disable the feature at compile time, or (2) allow the selection to occur at run time, as happens in the current program.[1]

Capturing feature selection. To solve the problem with edicts, we must capture the variation point within a design pattern, which is already done in this example. In addition, we must modularize the ways in which binding decisions—as opposed to *binding-time* decisions—are made. Concretely, we must modularize not only the binding sites of a feature but also the parts of the program that interact with the design pattern to select features. We refer to those parts of the program as *clients* of the pattern, and in our example, the client is the main program.

[1] One could use a general-purpose program specializer to achieve the desired effect in this example by specializing with respect to `args`. For large programs, performing specialization requires sophisticated dataflow analysis between feature-configuration and feature-use sites, which due to imprecision may not achieve complete optimizations. Edicts, conversely, are intended to support *straightforward* analyses by programs and people.

```
public interface Message {
    public void printThis(String s);
}
public class ConcreteMessage implements Message {
    public void printThis(String s) {
        System.out.print(s);
    }
}
```

```
public class QuoteDecorator implements Message {
    protected Message m;

    public QuoteDecorator(Message m) {
        this.m = m;
    }
    public void printThis(String s) {
        m.printThis("‘‘" + s + "’’");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Message msg = new ConcreteMessage();
        if ((args.length > 0) && (args[0].equals("1")))
            msg = new QuoteDecorator(msg);
        msg.printThis("Hello");
    }
}
```

Figure 5. Implementation of the example message-display program. Message quoting is an optional feature, bound at run time.

The `Main` class in Figure 5 has inlined logic—the `if` statement—to perform feature selection. The next step in applying edicts to this example, therefore, is to refactor the main program so that feature selection occurs where it can be stably identified by an AspectJ pointcut expression. Figure 6 shows the result, in which the feature selection logic has been extracted to a method called `makeMessage`. This method is not defined in the `Main` class. Rather, it will be introduced by an edict in order to implement a desired binding time. If there were several clients, it would be necessary to refactor all of them in a similar manner.

Implementing run-time binding. We are now ready to implement our edicts. The left-hand side of Figure 4 illustrates how our edicts will affect the classes within our application. The diagram shows the edict hierarchy, and the semi-circular dual arrowhead represents the application of advice and ITDs. The arcs connecting the edict hierarchy with the `ConcreteMessage` and `QuoteDecorator` classes, and the unfilled arc between these connectors, represents the edict's ability to "fill in" the pattern.

Given the refactored main program, it is straightforward to write an edict that implements late (run-time) feature binding. This edict

```
public class Main {
    public static void main(String[] args) {
        Main app = new Main();
        Message msg = app.makeMessage(args);
        msg.printThis("Hello");
    }
}
```

Figure 6. The refactored `Main` class

```
public aspect LateBindDecorators {
    public interface Client{};
    declare parents: Main implements Client;

    public Message Client.makeMessage(String[] args) {
        Message msg = new ConcreteMessage()
        if ((args.length > 0) && (args[0].equals("1")))
            return new QuoteDecorator(msg);
        return msg;
    }
}
```

Figure 7. An edict to select message-quoting at run time

is shown in Figure 7.[2] The edict declares a marker interface and uses it to identify all the clients of our message feature. Next, the edict defines an implementation of `makeMessage` that performs feature selection according to the run-time values of `args`, as was done in the original `Main` class of Figure 5.

Implementing compile-time binding. Two edicts are required to implement early (compile-time) feature selection: one that disables the message-quoting feature, and one that enables it. These are shown in the top two panels of Figure 8, and they are similar to the edict for late binding that was described above. If we had a wider variety of decorators, a programmer could easily write edicts for those as well (perhaps in a style supporting decorator composition, which is not shown in this simple example).

To produce a version of the message-printing program, a programmer includes exactly one of the late-binding, early-binding-disable, and early-binding-enable edicts in the compilation. These correspond to three unique software products with different combinations of features and binding times.

There are three additional and important points to note about our early-binding edicts.

First, if a product designer selects the `EarlyBindDisableDecorators` edict, it is nearly trivial for a static program analyzer to remove the unused `QuoteDecorator` from the compiled product. This is because there are no references to `QuoteDecorator` in any class that is reachable from `Main.main()`.

Second, the early-binding aspects in this example include dynamic checks to ensure that the command-line options, which controlled message-quoting in the original program, are consistent with the feature configuration of the current program. The need for such checks depends on how feature selection is expected to occur in a particular system. We could have chosen to ignore the command-line arguments, but this would create an obvious source of confusion if the user believes that the command-line options should still be effective. We discuss this issue further in Section 3.4.

Third, because an edict controls the implementation of a design pattern, it sometimes has flexibility in terms of mapping pattern-defined roles onto the concrete classes of an application. Said differently, an edict can often be implemented in several different

[2] Because this is a simple example, we choose not to implement the abstract aspect that is part of most edict hierarchies (Section 3.2).

```
public aspect EarlyBindDisableDecorators {
    public interface Client {};
    declare parents: Main implements Client;

    public Message Client.makeMessage(String[] args) {
        if ((args.length > 0) && (args[0].equals("1")))
            throw new RuntimeException();
        return new ConcreteMessage();
    }
}
```

```
public aspect EarlyBindEnableDecorators {
    public interface Client {};
    declare parents: Main implements Client;

    public Message Client.makeMessage(String[] args) {
        if ((args.length > 0) && (args[0].equals("0")))
            throw new RuntimeException();
        return new QuoteDecorator(new ConcreteMessage());
    }
}
```

```
public aspect EarlyBindEnableDecorators_v2 {
    public interface Client {};
    declare parents: Main implements Client;

    public Message Client.makeMessage(String[] args) {
        if ((args.length > 0) && (args[0].equals("0")))
            throw new RuntimeException();
        return new ConcreteMessage();
    }

    pointcut MessageMethodCall(String s):
        call(* *.printThis(..)) &&
        target(Message) && args(s) && within(Main);

    // Perform 'QuoteDecorator' behavior.
    void around(String s): MessageMethodCall(s) {
        s = "‘‘" + s + "’’";
        proceed(s);
    }
}
```

Figure 8. Edicts to bind *(disable or enable)* the message-quoting feature at compile time

ways to achieve the same effect, while addressing different implementation concerns. For example, the third panel of Figure 8 shows an alternative implementation of the edict that enables message-quoting at compile time. This version implements the message-quoting decorator as around advice, and in the process, makes the `QuoteDecorator` class dead. Reassigning roles becomes more difficult as the complexity of a program increases, of course, but our point is that an edict provides the modularity mechanism for making such changes. Choosing the "best" implementation of an edict is a matter of engineering; our technique permits a variety of edict implementations that may be tailored to different contexts.

3.4 Discussion

Based on our experience, we make some additional observations about the applicability and implementation of edicts.

Applicability. In this paper, we use edicts in combination with four patterns from Gamma et al. [9]: *Abstract Factory*, *Decorator*, *Factory Method*, and *Proxy*. We believe that edicts can be useful in combination with many other design patterns as well, to capture relationships whose binding times are variable across a product line. Chakravarthy's thesis [1] analyses the applicability of edicts to all of the patterns presented by Gamma et al.

Edicts and patterns. We found that design patterns were useful for implementing the different feature variation points that we encountered in this work. The patterns we applied helped us modularize the implementations and expose join points that we needed in order to control binding-time decisions via edicts.

That said, we do not claim that patterns are sufficient or necessary for implementing all variation points in software product lines. For example, features that are logically part of an application's execution environment—e.g., automatic object persistence and profiling support—are examples that do not fit with our notion of edicts as "targeted aspects." Conversely, the "interfaces" to some features are simple enough that a programmer does not to need patterns to encapsulate their bindings. In these cases, simpler application program interfaces (APIs) could be targeted by edicts to implement binding-time flexibility. What is important is the ability to identify binding sites in the code, in a modular fashion. For the features we studied, patterns provided the modularity we needed.

Dynamic configuration. When a variation point may be bound early or late, software must be designed carefully so that attempts to dynamically select a feature do not conflict with choices that were made statically. We mentioned this problem in Section 3.3. For users, the ultimate solution is communication: providers must make the capabilities of their products clear to users. Even when users know what to expect, however, our experience highlights an implementation problem: it can be difficult to fully modularize the code that leads to a dynamic feature selection.

For example, features may be selected based on data read from a user's configuration file. The flow of information from the file to the configuration point can be arbitrarily complicated, and therefore, it can be difficult to modularize the entire program "slice" that leads to a particular binding decision.

In practice, therefore, when transforming a variation point into one that supports both early and late binding, we take a pragmatic but safe approach. We leave configuration parsing code intact, and find a program point at which we can insert a run-time check to ensure that an early-bound feature choice is compatible with any incoming dynamic configuration data. For example, we might use a `set` join point to check values that are written into an object that holds data read from a configuration file. Ideally, this point is reached early in the program (during system initialization) so that conflicts are detected and handled as quickly as possible. The important issue is that edicts allow us to support both flexible feature binding times and consistency. A well-designed edict can introduce both early bindings—supporting optimizations—and dynamic checks when necessary—supporting safety—to ensure that products with early-bound features are used correctly.

3.5 A method for utilizing edicts

In this section we describe a general procedure for introducing binding-time flexibility using edicts into a software product line. Figure 9 illustrates the workflow, consisting of the steps described below. The steps are shown in the figure as circled numbers.

Step 1: Identify and characterize the variation points. In the initial stages of software development, a feature model helps to define software behavior and guide the discovery of variation points. Once variation points are identified, a designer must analyze the behaviors of those points. If applicable, for each point, the designer chooses one or more design patterns to capture the behavior. The selected patterns clearly characterize the behavior by identifying the participants and their roles and interconnections.

Step 2: Express domain concerns using binding times. Variability arises from the need to address different domain concerns, and some concerns can be addressed by choosing the times at which features are bound. In some cases, a variation point may need a flexible binding time in order to address different concerns across different products. Variation points exhibiting this hierarchy are picked out and examined. For every such variation point, the software designer reexamines the design pattern that characterizes the point's behavior. The selected pattern must be flexible enough to allow a product assembler to be able to make the choice to add/remove variants or effect static/dynamic behavior at the variation point as late as possible in the development process.

Step 3: Implement the pattern. Design patterns can be implemented in different ways. The traditional approach outlined by Gamma et al. [9] implements the pattern within the participants' code. An alternate approach is described by Hannemann et al. [11]: their technique encapsulates the roles of the participants and abstracts the pattern from its application. (Garcia et al. compare the benefits of these approaches [10].) Either approach can be employed in conjunction with edicts. Often, parts of the pattern implementation are deferred to edicts, which are written next.

Step 4: Introduce binding-time variability using edicts. We suggest that as an initial step, the design pattern implementation support only run-time binding. This helps to clarify and direct the process of introducing binding-time variability; basically, it helps identify and if necessary implement a parameterized method in which configuration is carried out. Based on the knowledge gained in step 2, the designer writes pointcuts to intercept the run-time binding decisions. As this pointcut will be utilized by both early- and late-binding edicts, the pointcut is written in an abstract aspect. The original pattern is refactored to move the binding time decision code into the late binding edict. Similarly, the early-binding edict is written to enforce static assertions.

Step 5: Optimize the resulting configured products. Edicts assist code optimizers by fostering static information in cases involving early binding times. As an example, consider a design pattern involving many possible types of participants. In the implementation of the pattern, the participants can usually be implemented as discrete classes. In situations involving compile-time binding, only one of the participants may be chosen. This choice is enforced by returning an object of the participating class at the variation point. This kind of explicit choice, enforced by an edict, is the static information that helps guide a code optimizer.

The process described above has been useful to us for incorporating binding-time flexibility into product-line software. It is our belief that the technique can improve safety in deployed products as well. Additional safety comes from the possible use of a static configurability checker to perform checks on configurations. For domains favoring early binding times, this checking would help evaluate the configuration against the domain constraints. By factoring out unsupported features, the checker would be able to provide efficient product customization.

4. Case Study: A JacORB Product Line

We now describe our application of edicts to four variation points within JacORB [13], an open-source CORBA implementation written in Java. We used JacORB version 2.2.3 in all our experiments. Ordinarily, JacORB runs on the Java SE (Standard Edition) platform. We created a product line by modifying JacORB to run on the embedded Java ME (Micro Edition), specifically the Connected Device Configuration (CDC), platform. We were unfamiliar with JacORB's implementation when we started this work, but we had significant expertise with CORBA implementations in general.

JacORB running on the Java ME-CDC platform provides the middleware advantage: a generic interface that developers can use for writing cross-platform applications. However, simply providing a generic interface is not enough. The Java ME-CDC platform is designed to run on devices with relatively small amounts of memory (512 KB to several MB). JacORB and its applications must fit within this constrained memory. To facilitate this, we used edicts

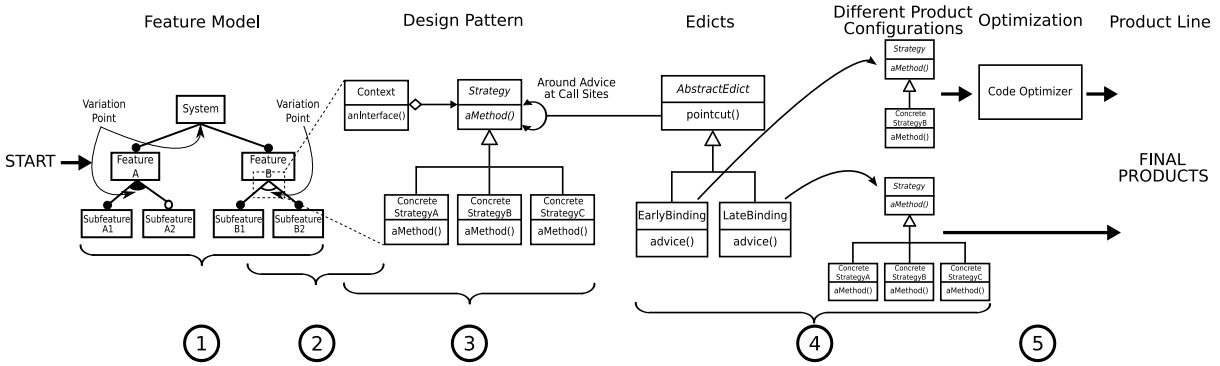

Figure 9. A method of using edicts to implement binding-time flexibility

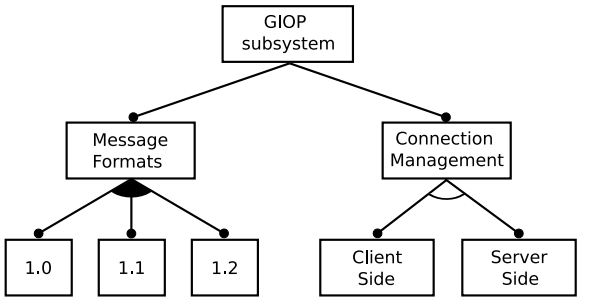

Figure 10. Feature model of a GIOP subsystem

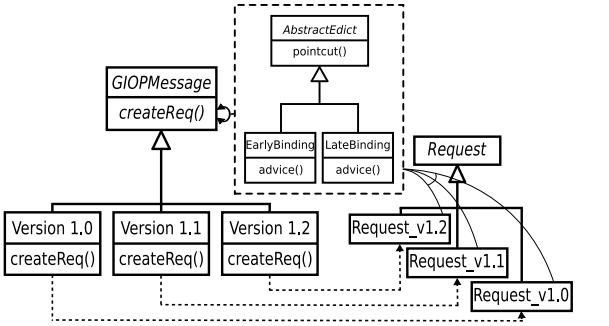

Figure 11. The *Abstract Factory* for GIOP `Request` messages

to implement flexible binding times for four selected features in JacORB. The edicts provided the required balance between generality and platform-specialization in our JacORB product line.

Following the method of Section 3.5, we identified variation points in JacORB that benefit from binding-time variability between the Java SE and Java ME-CDC platforms. We chose feature selection points within of the GIOP, security, and portable interceptor subsystems in JacORB. We then (re-)implemented those variation points using design patterns. Finally, we wrote edicts in AspectJ to manipulate binding times at those points.

To configure JacORB for Java ME-CDC, we used edicts to enforce compile-time bindings at our selected variation points—effectively "hard-wiring" certain feature choices. In this configuration, edicts allowed us to optimize JacORB for the memory-constrained platform. For Java SE, on the other hand, we compiled JacORB with edicts that enabled run-time feature selection.

The rest of this section describes how we implemented binding-time flexibility for our selected variation points within JacORB. Section 5 describes the benefits that we obtained.

4.1 GIOP version selection

CORBA middleware uses the General Inter-ORB Protocol (GIOP) to encode messages between client and server programs. JacORB supports three versions of GIOP: 1.0, 1.1, and 1.2. There are substantial differences between the versions of the protocol. GIOP also leaves policy decisions, such as choice of connection management strategy, open at the client and server sides. We modeled these choices in a feature model of the GIOP subsystem, depicted in Figure 10. We consider GIOP version selection below, and connection management in Section 4.2.

As suggested by our feature model, not all versions of GIOP need to be supported when JacORB is run on a limited-memory device. JacORB and its clients can run fine with just GIOP 1.0, which is supported by nearly all CORBA systems.

The original implementation of the GIOP-version selection logic in JacORB used `if-else` constructs to select among GIOP versions. This selection logic was duplicated across six different classes—obviously inefficient and not amenable to change or feature subsetting. We re-implemented the GIOP version selection code using the *Abstract Factory* design pattern. We introduced new class and edict hierarchies as part of our pattern implementation, and refactored the above six locations with calls to the pattern. Figure 11 summarizes our GIOP *Abstract Factory* and the edicts for controlling binding time.

The *Abstract Factory* pattern supports the creation of related objects without specifying the concrete classes [9]. Each GIOP version (1.0, 1.1, and 1.2) is represented by a factory class. These factories are used to construct different version-specific kinds of messages: request, reply, locate-request, and so on.

We used edicts to implement binding-time behavior at binding sites in the *Abstract Factory*. The binding sites are the calls to `createReqMsg()` (in the six refactored classes) that construct `Request` messages. Our late-binding edict inserts advice that implements dynamic binding: the decision about the GIOP version for the `Request` message is made dynamically using a run-time argument. We wrote three early-binding edicts, one for each version of GIOP. Each uses advice that creates a `Request` for the version of GIOP that was chosen at compile time by the product assembler.

4.2 GIOP client-server selection

In JacORB, client and server applications use different functionality within the GIOP subsystem. An ORB installed on a mobile embedded device is most likely to function as a client, connecting to servers running on resource-rich machines.

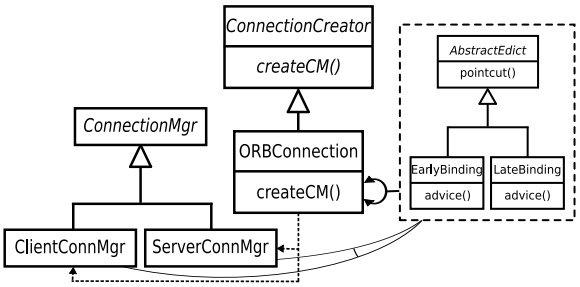

Figure 12. The *Factory Method* for GIOP connection management

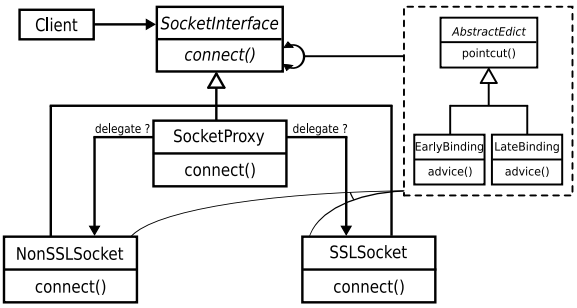

Figure 13. The *Proxy* pattern for the security policies

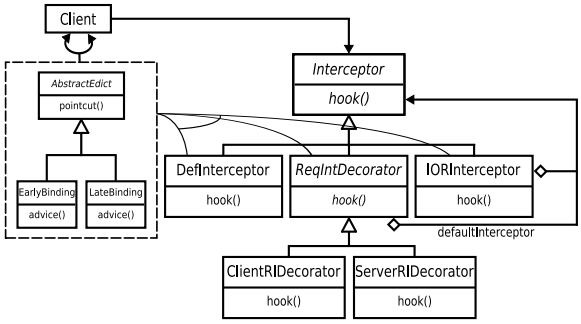

Figure 14. The *Decorator* pattern for Portable Interceptors

The original implementation of JacORB did not cleanly separate client and server behaviors within its implementation—an ORB could always function as both client and server. We refactored JacORB's GIOP subsystem to separate client and server operations. More specifically, we used the *Factory Method* design pattern and edicts to enable the creation of client-only, server-only, and client-server ORBs. Figure 12 illustrates the pattern and our edicts.

Using *Factory Method*, we described JacORB functionality such as connection management and communication message types in interfaces. Client and server classes then provided their own implementations of the interface methods. The parameterized factory methods are the binding sites where edicts manipulate the binding times of client and server features. Our late-binding edict allows the ORB to dynamically behave as either a client or server. We used early-binding to create a client-only ORB for the Java ME-CDC version of JacORB.

4.3 The security subsystem

JacORB provides a generic socket interface for communication. Sockets can either be secure (using SSL) or insecure. Secure communication is desirable, but comes with heavy requirements on memory and computation power. We decided to forgo secure sockets for the Java ME version of JacORB in favor of smaller static code footprint.

Secure communication in JacORB is provided by two separate implementations: the IAIK Java Cryptography Extension (IAIK-JCE) package and the Java Secure Socket Extension (JSSE) package. Both implement the SSL protocol and provide data encryption, authentication, and message integrity. Non-secure communication in JacORB is also implemented in two ways: a default implementation using simple Java sockets, and another that ensures all socket port addresses fall within a specific range.

In the original JacORB, the different socket types were implemented as classes that implement the generic socket interface. JacORB-based applications chose a communication strategy by referring one of the four concrete socket classes. Although this design was modular, it was not very abstract. Furthermore, the choice of a socket class was generally made by reading a value from a configuration file at initialization time, when JacORB started up. The unavailability of configuration data before initialization time made it difficult to get rid of unused socket classes.

We re-implemented the interface to sockets using the *Proxy* design pattern. As shown in Figure 13, applications now use the `SocketInterface` to make socket connections. An application is provided with a `SocketProxy` object. Our late-binding edict allows an application to pick, at run-time, either secure or insecure communication. An early-binding edict advises all clients of the `SocketInterface`, bypassing the proxy, and provides applications with the non-secure, default socket implementation. The edict's static choice in the Java ME version of JacORB causes all the other socket implementations to be dead code.

4.4 Portable interceptors

Portable Interceptors (PIs) are a framework for introducing new behavior into an ORB, e.g., tracing or security. PIs can intercept messages between clients and servers, modify the communication, and thereby change the behavior of a system. An example of this kind of usage would be to perform load balancing among several servers, transparently to the clients that issue requests.

There are different types of PIs: client, server, and IOR. The first provides interception points for client ORBs, and the latter two provide interception points for server ORBs. The original JacORB provided support for all three kinds of PIs—supporting only client-side or server-side PIs was not a configurable option. We decided to implement this choice, since we wanted to support only client-side PIs in our Java ME version of JacORB.

We found that all three kinds of interceptors could be implemented as decorators of the generic `Interceptor` interface. Figure 14 summarizes our new implementation, which applies the *Decorator* pattern to Portable Interceptors. The `DefInterceptor` class provides a default implementation of the `Interceptor` interface. Decorator classes implement the client, server, and IOR interceptors. To provide early binding, we wrote edicts that statically bind support for the client interceptor only. In the case of late binding, a client chooses the decorator that it needs to use.

5. Evaluation

In applying edicts to JacORB, our goal was to create a software product line that allows us to tailor the middleware for memory-constrained, Java ME-CDC devices. Because edicts control the binding times of features, the primary metric for evaluating edicts in this context is the achieved reduction in the static code size of the middleware. (Edicts do not generally affect how features operate internally, and thus, have minimal impacts on dynamic metrics such as execution time.) The use of design patterns and edicts does

UU IR Author Manuscript

not produce static optimizations by itself. Rather, as we show, our technique allows static code optimizers to be much more effective.

5.1 Experiment setup

We implemented edicts in JacORB 2.2.3 as described in Section 4. To configure JacORB for Java ME-CDC, we included early-binding edicts in the compilation of the middleware. These select GIOP 1.0, default (non-secure) sockets, and client-side features only. To configure the middleware for Java SE (desktops), we apply late-binding edicts, which allow feature selection at run time. We compile the middleware with `ajc` version 1.5.3.

We used ProGuard [16], version 4.0 beta, as our static program optimizer. ProGuard implements three primary transformations, which it calls *shrinking*, *optimization*, and *obfuscation*.

- *Shrinking* is based on a whole-program, inter-procedural, flow-insensitive, and context-insensitive analysis that identifies unused classes, methods, and fields within a program. Starting from a set of roots, the shrinker finds all the program elements that may be (directly or indirectly) referenced at run time. Elements that cannot be referenced are then removed, thereby reducing the program's static code size.
- *Optimization* implements a family of bytecode transformations intended to make programs smaller (and faster, in general). Many transforms, such as dead-code elimination, are driven by a (context-insensitive) partial evaluator for methods, which tracks values through local variables and the stack. Other optimizations, such as inlining methods and removing write-only fields, are driven by whole-program control-flow and dataflow analyses. ProGuard normally runs its shrinker before optimization, to save analysis time. It also runs the shrinker afterward, to remove elements that the optimizer made—or revealed to be—dead.
- *Obfuscation* replaces program symbols—e.g., class, method, and field names—with shorter ones, without otherwise affecting the program. In our experiments, we disabled obfuscation because it is unrelated to the bytecode analyses that are affected by edicts.

In summary, ProGuard implements a set of program analyses that are reasonably sophisticated but somewhat limited in terms of their ability to discover and exploit static information within a Java program. In our results below, we show how edicts allow ProGuard to be more effective, by putting optimization opportunities within the reach of its shrinking and optimization passes.

We applied ProGuard in two ways to highlight the benefits of edicts in different situations. First, we used ProGuard to specialize JacORB in a way that preserves the entire API of the middleware. This describes the case in which the middleware is the "delivered product," intended to support many (possibly unknown) applications. We refer to this as the *API-Preserving JacORB*. Second, we used ProGuard to specialize JacORB for a particular application; this represents the case in which the application is the delivered product. We refer to this as the *Application-Specific JacORB*.

For the application-specific case, we wrote a simple client program that fetches and displays text messages from a server. We ran this application on an emulator of a Sony Ericsson P990 series cell phone [19]. The P990 supports Java ME-CDC; we emulated a phone model with 1.5 MB of application memory.

5.2 Subsystem-level results

Each of the four subsystems that we modified is contained in its own Java package hierarchy. Most of our work using patterns and edicts was localized in these packages; the remaining refactoring occurred in the classes that used the features we modified.

Figure 15 shows how the static sizes of these subsystems are affected by patterns and edicts, *prior to* shrinking and optimization by ProGuard. Here, static size is the sum of the sizes of the class files in the compiled subsystem; this includes classes that are dead. Each

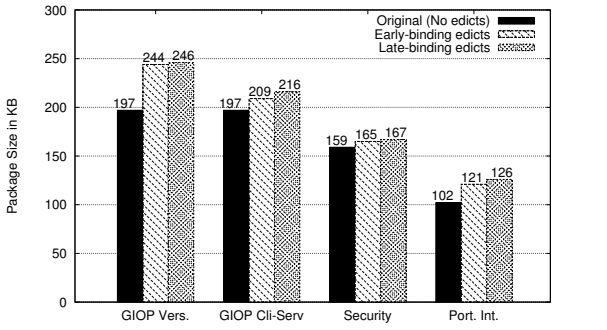

Figure 15. Per-subsystem static code sizes, before optimization

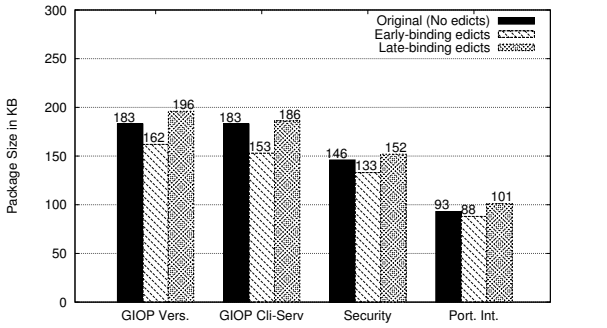

Figure 16. Per-subsystem static code sizes, after optimization

subsystem is measured in three configurations: the original JacORB implementation, our modified version using an early-binding edict, and our modified version using a late-binding edict.

As the graph shows, we observed an increase in the package sizes for the early- and late-bound edict implementations across all subsystems. We looked at the bytecode of the compiled classes to understand the cause of this increase. Overall, we were able to attribute this to two factors: (1) re-implementing the subsystems with patterns and aspects introduced new classes into each subsystem; and (2) `ajc` adds bytecodes to perform aspect weaving. The package sizes for the early- and late-binding edict versions are roughly equal. This happens because, although an early-binding edict enforces a static choice, this does not cause dead code to be removed.

We are now in a position to apply ProGuard to the three versions of each JacORB subsystem. Before optimizing the original JacORB versions, we modified JacORB's dynamic feature-selection code so that it would make hard-wired feature choices equivalent to those made by our early-binding edicts. This introduced information into JacORB that could conceivably be used by ProGuard to optimize the original subsystems.

Figure 16 shows the static sizes of the subsystems *after* shrinking and optimization by ProGuard. Here, the benefits of early-binding edicts become apparent. Across all subsystems, the early-bound versions are smaller than the optimized, original versions. With late binding, the "edictized" subsystems are comparable to—but still larger than—the optimized, original subsystems.

Despite the increases before applying ProGuard, the early-bound edicts enable better dead-code elimination and whole-program optimization. Most of these size reductions are attributable to the specialization of the patterns and the removal of classes that represent features not selected by the early-binding edicts. In the original implementation, ProGuard was unable to exploit the static feature-selection information that we provided: it could not prop-

UU IR Author Manuscript

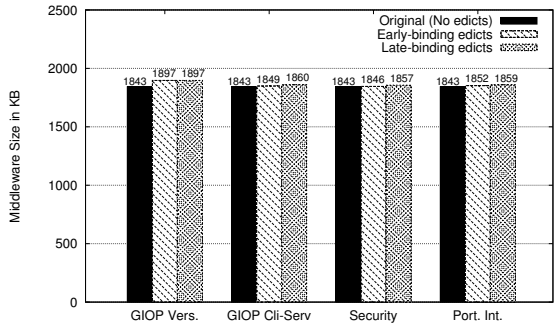

Figure 17. JacORB static code sizes, before optimization

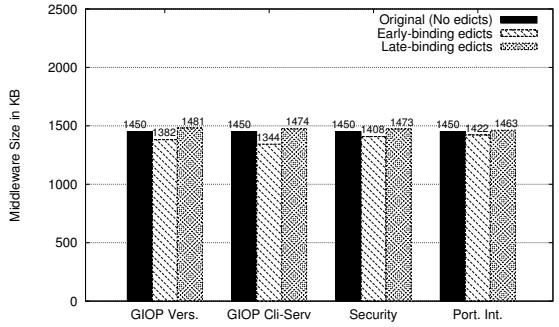

Figure 18. API-Preserving JacORB: code size after optimization

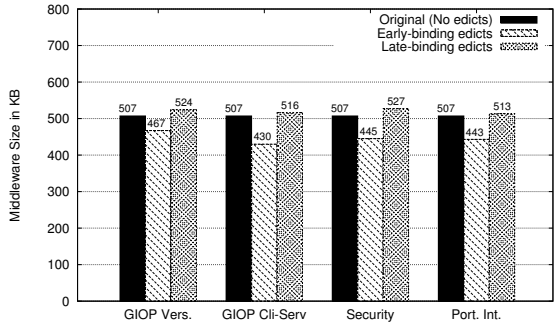

Figure 19. App.-Specific JacORB: code size after optimization

agate our hard-wired feature choices from where they were set to where they were used. To have succeeded, ProGuard would have needed to perform a complex, whole-program, inter-procedural dataflow analysis capable of tracking values through fields in heap objects. In contrast, when early-binding edicts are used, the structure of the program (woven bytecode) makes the desired optimizations obtainable via much simpler program analyses—like those implemented by ProGuard's shrinker and optimizer.

5.3 Whole-middleware results

In this section we again measure the effects of individual edicts, but in the context of the complete middleware. Figure 17 shows the static code size of JacORB—the size of all class files in the middleware—when we introduce edicts for individual subsystems. For example, for the security subsystem results, we applied edicts only to that subsystem while leaving the other subsystems in their original un-edictized forms. The size of the original middleware (without any edicts) is of course the same across all four sets of bars. The data show that the size increase of any one unoptimized edict is small, relative to the total size of JacORB.

As described in Section 5.1, we explored the results of optimizing JacORB in two ways. First, we ran ProGuard to produce an *API-Preserving JacORB*. We directed ProGuard to retain all of the functionality that was part of the core ORB's API. This yielded a generic middleware system that could be reused to write many applications. Second, we created an *Application-Specific JacORB* by directing ProGuard to retain only the code that was used by our sample Java ME-CDC messaging application. This produced a minimal middleware system, like one that a vendor might incorporate into a specific application product. In both cases (and as described previously), before optimizing the original JacORB, we inserted static feature-selection data into the part of JacORB that is normally responsible for configuring features dynamically. Thus, ProGuard had access to equivalent static information when optimizing the original and early-bound versions of the middleware.

API-preserving JacORB. Figure 18 shows the results of applying shrinking and optimization to obtain an API-Preserving JacORB. Again we show the effect of enabling one edict at a time, and we show the size of the optimized, original middleware in each group of bars. All early-bound JacORB versions show smaller code sizes across all subsystems. Moreover, in comparison to the subsystem-level results shown previously in Figure 16, the absolute code-size savings due to each early-binding edict are larger. These improvements at the middleware level can be attributed to a "cascade effect" caused by the early-binding edicts in the optimization process. The edicts specialized the pattern and thereby exposed static information within the clients of the pattern. This in turn allowed ProGuard to perform its optimizations more effectively.

As an example, consider the *Factory Method* modification in the GIOP client-server subsystem. There were four pointcuts in the early-binding edict used with the *Factory Method*. These pointcuts advised five different classes that used the *Factory Method* at eight different locations. Three of the classes that were advised were classes where core ORB processing occurred. By making static information available at these call sites, whole sections of code—methods and classes devoted to server-side processing—became dead and were removed by ProGuard.

In the case of late-binding edicts, the sizes of the API-preserving JacORBs are again slightly larger but very close to the size of the original-code JacORB.

Application-specific JacORB. Figure 19 shows the static code size results after optimizing JacORB with respect to our sample application to produce a custom version of the middleware. When compared with the unoptimized JacORB implementations from Figure 17, we observe massive savings in code sizes. The original JacORB version shrunk to 507 KB from 1843 KB, a savings of 72.5%. However, in all cases the early-binding edict versions of JacORB achieved even greater savings, as shown. The four application-specific, early-edict versions of the middleware are 7.8%, 15.2%, 12.2%, and 12.6% smaller than the application-specific, original JacORB.

The code sizes of the optimized JacORBs with late-binding edicts are again similar to the sizes of optimized, original JacORB.

5.4 Combined early-bound configurations

Finally, we combine the modifications that we performed for the four individual subsystems and choose the early-binding edicts in all subsystems. As mentioned previously, we use early binding to select GIOP version 1.0, a client-only GIOP connection, no SSL sockets for communication, and client-side Portable Interceptors. We compile JacORB with all of these edicts selected. Finally, we

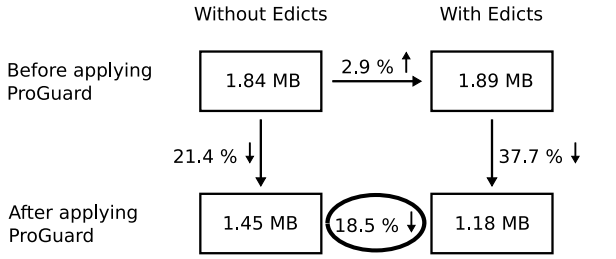

Figure 20. Early-bound edicts for API-preserving JacORB

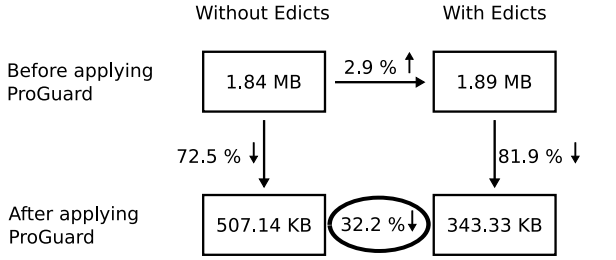

Figure 21. Early-bound edicts for application-specific JacORB

run ProGuard to create both an API-preserving JacORB configuration and an application-specific JacORB configuration.

Figure 20 summarizes the results for the early-bound, API-preserving version of JacORB. The top-left box in the diagram shows the static code size of the original JacORB: we did not insert patterns or aspects into this implementation, and all features are bound either at initialization time or at run time. The size before applying ProGuard is 1.84 MB. After refactoring the four subsystems with edicts, we enabled early-binding of features in each of them. The size of this version of JacORB, before ProGuard processing, is 1.89 MB: a 2.9% increase in size due to the edict refactorings. The top-right box in the diagram represents this refactored, early-bound implementation. After processing both the original and early-bound JacORB versions with ProGuard, the API-preserving ORBs have sizes of 1.45 MB and 1.18 MB respectively. Comparing the sizes in the lower two boxes shows that for an API-preserving ORB, there is an overall 18.5% reduction in static code size for the early-bound JacORB as compared to original JacORB. This result is the approximate sum of the individual improvements observed in the four subsystems under a similar setup.

Figure 21 shows our results for an early-bound, *application-specific* version of JacORB. Following a similar analysis as above, for an application-specific version of JacORB with early-binding enabled in all four subsystems, after processing with ProGuard, we derived a 32.2% improvement in static code size as compared to the original, late-bound JacORB implementation.

6. Related Work

Binding time has long been known to be an important concern in product line software development [2, 4, 5, 21], and it continues to be an active focus of research. Svahnberg et al. [22] provided a classification of many variability realization techniques, including techniques based on aspect-oriented programming, and cataloged the techniques according to the binding times that they support. In their classification system, the use of edicts to provide binding-time flexibility falls in the category of "Code Fragment Superimposition." This category encompasses a variety of aspect-like techniques that overlay new features onto an existing software system. Edicts, however, are more specific: edicts are not intended to introduce new features per se, but rather to control the selection and binding times of existing features. To achieve this, we realize feature variation points using design patterns and then superimpose the edict "code fragments" on those pattern implementations only. Thus, edicts are intended to be stable under program evolution, in the spirit of recent work on stable programming interfaces for aspect-oriented design [20].

Binding-time flexibility was explored previously in Koala [23], a system for consumer electronics software. Whereas we have described an idiom for implementing binding-time flexibility, the Koala component model provides a specific construct for controlling the binding times of features within software products. In Koala, a product is an assembly of components that are implemented in C and connected via an explicit linking graph. A link from one component's export to another component's import can be routed through a construct called a *switch*, which directs function calls from a "client" component to one of many "service" components that provide implementations. Thus, a switch encapsulates a variation point. A single switch can also be used to route multiple client-service connections in a coordinated manner. The setting of a switch can be determined at run time or at compile time; Koala applies a partial evaluator at compile time to optimize switched connections as far as possible. Koala's switches are therefore quite similar, both in intent and functionality, to the implementations of variation points that we create using design patterns and aspects. However, whereas Koala switches are provided by a specialized component model for C, our technique is realized via standard programming idioms and AOP for Java. In addition to being widely applicable, our approach makes it possible for programmers to readily perceive variation points as implementations of patterns—information that may be obscured when a pattern is implemented as linkages in a component assembly [6].

Edicts facilitate optimization in design pattern implementations that correspond to variation points. Partial specialization [3] and specialization patterns [18] are techniques intended to optimize programs and design pattern implementations more generally. Partial evaluators follow the technique of reduction by *proof*: a partial evaluator uses knowledge provided by a programmer or a static analysis to evaluate program fragments and replace them with their results. This analysis process can be quite complex and time consuming; moreover, the transformations that are performed by a general partial evaluator can be difficult for a programmer to predict and understand. Edicts, however, provide useful information in a human-understandable manner, and the effect of an edict can replace general-purpose analyses for variation points and/or make subsequent analyses more predictable. For example, an early-binding edict can save the expensive computation an evaluator would have to perform to reach the same result that is stated directly by the edict. In this way, the use of edicts is complementary to the use of general partial evaluators. Specialization patterns use partial evaluation to specialize implementations of design patterns; overheads introduced by abstraction are removed. Edicts work with design patterns in a similar but human-directed way, to express feature bindings in a way that is obvious to software developers.

We created a middleware product line by refactoring one of its subsystems to produce a range of configuration options. Middleware specialization and product lines are an active area of research. For example, the FOCUS toolkit [15] is designed to specialize TAO-based middleware products. FOCUS uses annotations in the code and a post-processor to perform specializations. Every annotation has a start and end; method calls that fall with an annotation section may be replaced by optimized calls. This process is effective, but requires that annotations be placed at scattered points in the middleware implementation. Our technique is more closely tied to the code: rather than annotate points for specialization, we

use design patterns to make specialization points identifiable by AspectJ pointcuts. We benefit from AspectJ's support for general-purpose AOP, whereas the developers of FOCUS had to implement specialized tools to achieve their goals.

Refactoring middleware using aspects has been addressed by Zhang et al. [24, 25], and they propose a *post-postulated architecture* [24] for middleware to deal with customization. Our approach is complementary to theirs: whereas they focus on variations of feature assemblies, we focus on varying features' binding times. Edicts leverage the benefits of design patterns and aspects to improve the configurability of feature-based software product lines.

7. Conclusion

We have presented and evaluated a new technique—a variability mechanism—for implementing the binding sites of software features that require flexible binding times. Binding time flexibility is important in software product lines where some products are fully featured and others—due to resource constraints or other reasons—implement a subset of the full functionality. Our variability mechanism uses design patterns in combination with targeted aspects, called edicts, in order to support flexible binding times for features. When a product assembler chooses to bind a feature at compile time, our approach results in code that supports compile-time analysis and optimization. In short, edicts help to ensure that opportunities for static optimization are not lost.

We have described a method for introducing edicts into existing software, and we have demonstrated the utility of edicts by using them to create a middleware product line based on JacORB. Early-binding edicts were successful in creating stripped-down versions of the middleware that were smaller than the original middleware. By applying early-binding edicts to just four variation points within JacORB, we created middleware configurations that were consistently smaller—up to 32.2% smaller—than the original JacORB software optimized without edicts.

Acknowledgments

We thank David Johnson and the anonymous AOSD reviewers for their insightful comments on drafts of this paper. Their suggestions helped us to improve this paper greatly.

References

[1] V. Chakravarthy. Adaptive product line design using aspects and design patterns. Master's thesis, University of Utah, 2008.

[2] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2001.

[3] C. Consel, L. Hornof, F. Noel, J. Noye, and N. Volansche. A uniform approach for compile-time and run-time specialization. In *Selected Papers from the 1996 International Seminar on Partial Evaluation*, pages 54–72, Dagstuhl Castle, Germany, Feb. 1996. Springer-Verlag, Berlin, Germany.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison Wesley, 2000.

[5] E. Dolstra, G. Florijn, and E. Visser. Timeline variability: The variability of binding time of variation points. In *Proc. of the 2003 Workshop on Software Variability Management (SVM)*, pages 119–122, Gronigen, The Netherlands, Feb. 2003.

[6] E. Eide, A. Reid, J. Regehr, and J. Lepreau. Static and dynamic structure in design patterns. In *Proc. of the 24th International Conf. on Software Engineering (ICSE)*, pages 208–218, Orlando, FL, May 2002.

[7] C. Fritsch, A. Lehn, and T. Strohm. Evaluating variability implementation mechanisms. In *Proc. of the 2002 Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing (PLEES)*, pages 59–64, Seattle, WA, Nov. 2002.

[8] C. Gacek and M. Anastasopoulos. Implementing product line variabilities. In *Proc. of the 2001 Symposium on Software Reusability (SSR)*, pages 109–117, Toronto, ON, 2001.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[10] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *LNCS*, pages 36–74. Springer, 2006.

[11] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. of the 2002 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173, Seattle, WA, Nov. 2002.

[12] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and, Organization for Business Success*. Addison Wesley, 1997.

[13] JacORB. `http://www.jacorb.org/`.

[14] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Software Engineering Institute Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[15] A. S. Krishna, A. S. Gokhale, D. C. Schmidt, V. P. Ranganath, and J. Hatcliff. Context-specific middleware specialization techniques for optimizing software product-line architectures. In *Proc. of EuroSys 2006*, pages 205–218, Leuven, Belgium, Apr. 2006.

[16] E. Lafortune. ProGuard. `http://proguard.sourceforge.net/`.

[17] R. E. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *Proc. of the 19th European Conf. on Object-Oriented Programming (ECOOP)*, pages 169–194, Glasgow, UK, July 2005.

[18] U. P. Schultz, J. L. Lawall, and C. Consel. Specialization patterns. In *Proc. of the 15th IEEE International Conf. on Automated Software Engineering (ASE)*, pages 197–206, Grenoble, France, Sept. 2000.

[19] Sony Ericsson. Symbian OS Docs & Tools. `http://developer.sonyericsson.com/site/global/docstools/symbian/p_symbian.jsp`.

[20] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *Proc. of the 10th European Software Engineering Conf. and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 166–175, Lisbon, Portugal, Sept. 2005.

[21] M. Svahnberg and J. Bosch. Issues concerning variability in software product lines. In *Proc. of the Third International Workshop on Software Architectures for Product Families*, pages 146–157, Las Palmas de Gran Canaria, Spain, Mar. 2000.

[22] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software—Practice & Experience*, 35(8):705–754, 2005.

[23] R. van Ommering. Building product populations with software components. In *Proc. of the 24th International Conf. on Software Engineering (ICSE)*, pages 255–265, Orlando, FL, May 2002.

[24] C. Zhang, D. Gao, and H.-A. Jacobsen. Towards just-in-time middleware architectures. In *Proc. of the 4th International Conf. on Aspect-Oriented Software Development (AOSD)*, pages 63–74, Chicago, IL, Mar. 2005.

[25] C. Zhang and H.-A. Jacobsen. Resolving feature convolution in middleware systems. In *Proc. of the 2004 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 188–205, Vancouver, BC, Oct. 2004.

UU IR Author Manuscript